APPLICATION

FOR

UNITED STATES LETTERS PATENT

APPLICANT NAME: Makhervaks et al.

TITLE: RDMA COMPLETION AND RETRANSMIT SYSTEM AND METHOD

DOCKET NO.: FIS920030282US1

INTERNATIONAL BUSINESS MACHINES CORPORATION

CERTIFICATE OF MAILING UNDER 37 CFR 1.10

I hereby certify that, on the date shown below, this correspondence is being deposited with the United States Postal Service in an envelope addressed to: Mail Stop Patent Application, Commissioner for Patents, P.O. Box 1450, Alexandria VA 22313-1450 as "Express Mail Post Office to Addressee" Mailing Label No. EV 225 574 102 US

on December 2, 2003

Beverly Kehoe Shea
Name of person mailing paper

Beverly Kehoe Shea
Signature

12/2/2003
Date

RDMA COMPLETION AND RETRANSMIT SYSTEM AND METHOD

BACKGROUND OF THE INVENTION

1. Technical Field

[0001] The present invention relates generally to remote data memory access (RDMA) completion and retransmit systems, and more particularly relates to an implementation of RDMA completion and retransmit that maintains ordering between the ResponseOut and RequestOut channels.

2. Related Art

[0002] RDMA (remote data memory access) is a network interface card (NIC) feature that lets one computer directly place information into the memory of another computer. The technology reduces latency by minimizing demands on bandwidth and processing overhead. Traditional hardware and software architecture imposes a significant load on a server's CPU and memory because data must be copied between the kernel and application. Memory bottlenecks become more severe as connection speeds exceed the processing power and memory bandwidth of servers.

[0003] RDMA gets around this by implementing a reliable transport protocol in hardware on the RNIC (RDMA network interface card) and by supporting zero-copy networking with kernel bypass. Zero-copy networking lets the RNIC transfer data directly to or from application memory, eliminating the need to copy data between application memory and the kernel.

[0004] Kernel bypass lets applications issue commands to the RNIC without having to execute a kernel call. The RDMA request is issued from user space to the local RNIC and over the network to the remote RNIC without requiring any kernel involvement. This reduces the number of context switches between kernel space and user space while handling network traffic.

[0005] RDMA consumers (applications) uses message semantics to communicate. The data is posted for transmit in messages, received in messages and completion is expected to be reported in units of messages. TCP in its turn is a byte-stream oriented protocol, which is not aware of possible message boundaries of ULP data. Therefore, the task of translation of message-to-byte stream semantics falls on RDMA, and their offloaded implementation.

[0006] RDMA is a message-oriented ULP that uses TCP reliability services. RDMA adds another level of complexity to the mapping of message-oriented ULP to the byte-oriented TCP semantics. RDMA uses the same TCP connection to transmit messages posted by two independent sources:

- RequestOut - RDMA requests originated by local consumer; and
- ResponseOut - RDMA responses originated by RNIC, as a result of reception and processing of inbound RDMA Read Request sent by remote consumer.

[0007] RNIC interleaves RequestOut and ResponseOut messages when transmitting them through the same TCP connection. In addition to the byte-stream to the message-stream mapping, RNIC needs to preserve "transmit ordering" between messages from RequestOut and ResponseOut, during completion and retransmit processes.

[0008] Different RDMA requests (e.g., "Fence") and completion ordering rules (e.g., RDMA Read Request is completed when RNIC receives an RDMA Read Response) may suspend a transmit or completion process in RequestOut. This suspension however does not prevent ResponseOut from performing transmit and completion of RDMA Read Responses. Accordingly, in order to preserve independence between the RDMA request queue and RDMA response queue, a system is required to efficiently preserve order between the two queues.

[0009] One approach to resolve this issue would be to build a single request/response channel implemented, e.g., using a control structure (descriptors) and/or by maintaining a separate copy of the data. Such an approach has several disadvantages, including:

- Additional copy operations are required;
- More RNIC resources are needed to implement such an approach (the data/control is copied to the adapter memory);
- Lack of flexibility (adapter memory is a limited resource, which limits the number of RDMA messages that can be outstanding on the wire); and
- Enforcing completion ordering between RDMA Requests and RDMA Responses is more difficult.

[0010] Accordingly, a solution is required to address the above-mentioned problems.

SUMMARY OF THE INVENTION

[0011] The present invention addresses the above-mentioned problems, as well as others, by providing an efficient solution for the completion and retransmit of RDMA streams, without the need for an additional copy of data or of control structures to maintain ordering. In a first aspect, the invention provides a process for handling a completion request in an remote data memory access (RDMA) environment having a RequestOut channel and a ResponseOut channel, comprising: a descriptor list for each channel, wherein each descriptor list includes a message descriptor for each message in the channel; an update mechanism for updating a message length field in the message descriptor with a sequence number of a last byte in the message whenever a channel swap occurs between the RequestOut channel and the ResponseOut channel; an acknowledgement (Ack) completion system that examines values in a completion context and compares a sequence number of a next to complete message with a last acknowledged sequence number to determine if the message should be completed; and a read request completion system that performs completion of a read request.

[0012] In a second aspect, the invention provides a method for handling a completion process in a remote data memory access (RDMA) environment having a RequestOut channel and a ResponseOut channel, including performing an acknowledgement completion on each channel with the steps of: concluding that the processing of the completion process is finished if the sequence number of a next to complete message is invalid; concluding that the processing of the completion process is finished if the sequence number of the next to complete message is greater than the last acknowledged sequence number; and completing the message in the channel if the

sequence number of the next to complete message is less than or equal to the last acknowledged sequence number.

[0013] In a third aspect, the invention provides a method for locating a segment to retransmit for a retransmit request in a remote data memory access (RDMA) environment having a RequestOut channel and a ResponseOut channel, comprising: identifying a candidate message for both the RequestOut channel and the ResponseOut channel; selecting a message carrying the segment to transmit from the two candidate messages; and determining a location of a pointer descriptor that refers to a beginning of the segment to retransmit.

[0014] In a fourth aspect, the invention provides a system for handling a transmit process in a remote data memory access (RDMA) environment having a RequestOut channel and a ResponseOut channel, comprising: a descriptor list for each channel, wherein each descriptor list includes a message descriptor for each message in the channel; and an update mechanism for updating a message length field in the message descriptor with a sequence number of a last byte in the message whenever a channel swap occurs between the RequestOut channel and the ResponseOut channel.

BRIEF DESCRIPTION OF THE DRAWINGS

[0015] These and other features of this invention will be more readily understood from the following detailed description of the various aspects of the invention taken in conjunction with the accompanying drawings in which:

[0016] Figure 1 depicts a completion and transmit system for maintaining ordering in accordance with the present invention.

[0017] Figure 2 depicts a message descriptor in accordance with the present invention.

[0018] Figure 3 depicts a flow diagram of a completion algorithm in accordance with the present invention.

[0019] Figure 4 depicts an example of updating the sequence numbers in RequestOut and ResponseOut channels in accordance with the present invention.

[0020] Figure 5 depicts a flow chart showing an Ack completion operation.

[0021] Figure 6 depicts a flow chart showing an RDMA read request completion operation.

[0022] Figure 7 depicts a pair of examples for maintaining completion ordering in a RequestOut and ResponseOut channel in accordance with the present invention.

[0023] Figure 8 depicts three retransmit cases in accordance with the present invention.

[0024] Figure 9 depicts a pair of examples for maintaining retransmit ordering in a RequestOut and ResponseOut channel in accordance with the present invention.

DETAILED DESCRIPTION OF THE INVENTION

[0025] Described below is a system and method for implementing RDMA completion and retransmit. Within the RDMA protocol, "completion" is defined as the process of informing the ULP (upper layer protocol) that a particular RDMA operation has performed all functions specified for the RDMA operations, including placement and delivery. The completion semantic of each RDMA operation is distinctly defined. "Retransmit" is defined as the process of retransmitting data across a channel.

[0026] As described below, the present invention utilizes a double-channel implementation (i.e., one for ResponseOut and one for RequestOut). To effectuate such an implementation, the invention provides a methodology for preserving ordering between the two channels whenever completion or retransmit operations are required.

[0027] It is assumed for the purpose of this description that the reader has an understanding of the RDMA protocol and its implementation in an RNIC environment. The RDMA protocol is available on the Web at < www.rdmaconsortium.org/home>.

Background - RDMA Protocol Ordering Rules

[0028] RDMA protocol defines completion ordering rules for the messages transmitted via the RequestOut channel. These rules are outlined as follows:

[0029] All RDMA requests must be completed in order they have been posted by a consumer. RDMA has three types of requests:

(1) RDMA operations consisting of a single RDMA message (e.g., Send, Write). These operations are intended to transfer the data from a local to remote host, and consist of single RDMA message sent from the remote to the local host. Send and Write operations are considered completed when RNIC can guarantee that all TCP segments comprising those messages would be reliably transmitted to other end point of the TCP connection.

(2) RDMA operations consisting of several (generally two) RDMA messages, e.g., Read. These operations are intended to read the data from the remote memory. Such operations consist of two RDMA messages: Read Request and Read Response. Read Request is a message that is sent by a request originator to the data source. This message

carries all the information necessary to read the data from the remote memory, and build a response message. The response message, generated by the remote host, is a Read Response. This message carries the data that should be written to the requester memory, and also includes a description of the location to which the data should be written. Read operations are considered completed when the RDMA Read Response message has been successfully received by the Read initiator. An RDMA Read Response is not a separate RDMA operation (i.e., it is part of the Read operation), and therefore the message is considered to be completed when the transport layer has successfully transmitted it.

(3) Local RDMA operations (e.g., Bind, Fast-Register, etc.). These operations do not result in the transmitting of any data (i.e., TCP segment). Local RDMA operations are considered as completed when their processing is finished by RNIC.

Overview

[0030] Referring to Figure 1, a completion and retransmit system 10 for preserving ordering between the ResponseOut and RequestOut channels is shown. The ordering is maintained when either a "request for completion" or "request for retransmit" is issued using either a completion algorithm 24 or a retransmit algorithm 26. To facilitate the process, system 10:

(1) Provides separate descriptor lists 12, 14 for the RequestOut and ResponseOut channels 16, 18; and

(2) Performs transmit, completion and retransmit operations using information from those descriptor lists.

[0031] In this approach, each work queue entry (WQE) is processed (i.e., read) twice, first when transmitting the WQE, and second when completing the WQE. (Note that a retransmit operation would require an additional read of retransmitted WQEs).

[0032] RNIC keeps information for each RDMA connection needed to maintain the connection. This information is kept in a connection context 22, and in this embodiment, a set of connection context fields are used to facilitate implementation of the completion 24 and retransmit 26 algorithms. In an exemplary embodiment, context fields are indicated by *Context[Ch#]::FieldName*. The following context fields are relevant to the present invention:

1. *Context[Ch#]::PendingCompletionRequest* - bit used to signify that RNIC has a pending Ack or RDMA Read completion request to handle.

2. *Context[Ch#]::CompletedReadRequestsNum* - used to signify the number of completed read requests.

3. *Context[Ch#]::ReqOutLastCompletedMsgSN* - used to signify a sequence number (SN) of the last completed message in a request out channel.

4. *Context[Ch#]::ReqOutNextToCompleteMsgSN* - used to signify a sequence number (SN) of the next to complete message in a request out channel.

5. *Context[Ch#]::RespOutLastCompletedMsgSN* - used to signify a sequence number (SN) of the last completed message in a response out channel.

6. *Context[Ch#]::RespOutNextToCompleteMsgSN* - used to signify a sequence number (SN) of the next to complete message in a response out channel.

7. *Context[Ch#]::LastAckedSN* - sequence number of the last received ACK.

8. *Context[Ch#]::PendingReadRequest* - this bit indicates that while performing a completion operation, RNIC stopped on the RDMA Read Request, and must wait for reception and delivery of the RDMA Read Response by receive logic to complete this Read Request, and then proceed to the completion of any following RDMA Requests

9. *Context[Ch#]::PendingReadRequestsCount* - value used to signify the number of transmitted and not-completed RDMA Read Requests.

10. *Context[Ch#]::PendingReadRequestCompletion* - needs to use PendingReadRequest bit.

11. *Context[Ch#]::TotalPostedMsgCount* - signifies a number of posted and not transmitted messages.

12. *Context[Ch#]::TotalPendingMsgCount* - signifies a number of transmitted messages waiting for completion per connection.

[0033] To represent an RDMA message in a descriptor list, several descriptor types are used. One of them is a Message Descriptor, such as that shown in Figure 2. This descriptor starts each RDMA message, and carries message description and information about other descriptors used to represent the message. The following fields are relevant to the invention:

1. *MessageDescriptor::MessageLength* - used to carry either the message length or the Sequence Number.

2. *MessageDescriptor::SN/MsgLengthBit* - used to describe the contents of the *MessageDescriptor::MessageLength* field.

Completion Algorithm

[0034] The completion algorithm 24 in accordance with the present invention has two completion triggers:

(1) Reception of a TCP Acknowledge (Ack). This indicates that the transmitted data has been successfully received on another end point of a connection, and the RDMA operations (Sends, Writes) covered by this Ack are completed. The RDMA Read Response message, as a part of the Read operation, is completed when the Ack covering this message has been received.

(2) Reception and delivery of an RDMA Read Response. This indicates that a corresponding RDMA Read Request (or pending Read operation) is completed.

[0035] Pending completion request indications are held in connection context 22 including *Context[Ch#]::PendingCompletionRequest* (which signifies that RNIC needs to perform a completion operation; this is actually means that the Receive Logic received either the new TCP Ack, or received and delivered the RDMA Read Response, and Completion logic should figure out what operation has been completed, if any, and report completion of those operations to the consumer) and *Context[Ch#]::CompletedReadRequestsNum* (which contains the number of completed read requests). This field holds the number of received and delivered by Receive Logic RDMA Read Response Messages. This effectively means that Completion Logic can complete that many pending RDMA Read Messages posted by consumer. From the RNIC perspective, Completion Logic completed that many RDMA Read Requests, and still needs to report that completion to the consumer.

[0036] The overall methodology for handing these cases is shown in Figure 3. First, at step S1, *Context[Ch#]::CompletedReadRequestsNum* is checked. If the value is non-zero (step S2), this indicates that the connection has a pending Read Request completion. In this case, RNIC first performs a "read request completion" operation described below, and then performs an "Ack completion" (steps S3 & S4), as also described below, regardless of the *Context[Ch#]::PendingCompletionRequest* bit. If the *Context[Ch#]::CompletedReadRequestsNum* equals zero and the *Context[Ch#]::PendingCompletionRequest* bit is set (step S5), RNIC performs the Ack completion (step S4), as also described below.

Ack Completion

[0037] A received TCP Ack completes an RDMA request posted to the RequestOut and ResponseOut channels 16, 18. The major challenge of completion process is to follow the order in which messages from ResponseOut and RequestOut channels were transmitted. To preserve this order, RNIC updates the Message Descriptors using update mechanism 25 of selected messages to carry the Sequence Number (SN) of the last byte of the message. This update process, also referred to as a "write-back," is performed when RNIC transmits a last segment of the message. RNIC uses the *MessageDescriptor::MessageLength* field to carry either the message length or the Sequence Number. The *MessageDescriptor::SN/MsgLengthBit* describes the content of the *MessageDescriptor::MessageLength* field.

[0038] RNIC does not update all messages, but only those used after a channel has been swapped, i.e., only the first message is updated after swapping the channel.

When transmitting RDMA messages, RNIC interleaves RequestOut and ResponseOut messages. RNIC arbitrates between the ResponseOut and RequestOut channels, and selects one of the channels to transmit the next message. Channels are not swapped in the middle of a transmitted message. When a swap occurs between channels, update mechanism 25 updates the message descriptor with the SN of the next message in the channel after the channel swap. After a swap, the message descriptor is updated only for the first message in the new channel, and the associated SN is updated to the SN of the last transmitted byte of that message. As shown in Figure 2, the information kept in the MessageLength field (either MessageLength or SequenceNumber) is then used during completion and retransmit process.

[0039] An example of the updating process is shown in Figure 4. In this example, RNIC swaps channels four times. It starts with message #1 in the RequestOut channel, then swaps to the ResponseOut channel (message #2), and updates this message with the SN of the last byte of this message #2. After message #3, RNIC swaps to the RequestOut channel for message #4, which it updates. After message #5, RNIC swaps to the ResponseOut channel for message #6, which it updates. After message #7, RNIC swaps to the RequestOut channel for message #8, which it updates.

[0040] A write-back operation is performed only for connections utilizing both channels. If a connection uses only one channel, no write-back operation is performed.

[0041] RNIC holds several sequence numbers in connection context 22 for each channel: *Context[Ch#]::ReqOutLastCompletedMsgSN*, *Context[Ch#]::ReqOutNextToCompleteMsgSN*, *Context[Ch#]::RespOutLastCompletedMsgSN,* and

Context[Ch#]::RespOutNextToCompleteMsgSN, which carry a sequence number (SN) of the last completed and first not completed message in each channel, respectively. They are initialized to carry an initial connection sequence number, and are updated during completion operations.

Ack Completion Flow

[0042] Completion flow is implemented by RNIC looping through a series of steps separately for both the RequestOut and ResponseOut channels. This process is described below. Note that the sequence number of the last received ACK is indicated in *Context[Ch#]::LastAckedSN.*

The steps are outlined as follows with reference to the flow diagram of Figure 5.

Step 1 (S10): If the channel is waiting for completion of an RDMA Read Request (i.e., *Context[Ch#]::PendingReadRequest* is set), then RNIC ignores the completion request (S11). Note that this rule is not relevant for ResponseOut channel since the ResponseOut channel carries RDMA Read Responses only.

Step 2 (S12): If *Context[Ch#]::NextToCompleteMsgSN* is invalid, the processing of the completion request for this channel is finished (S13). This occurs when either no more messages have been posted, or a next posted message was not entirely transmitted.

Step 3 (S14): If *Context[Ch#]::NextToCompleteMsgSN* > *Context[Ch#]::LastAckedSN*, then the received completion request does not complete any message in that channel, and processing of the completion request for this channel is finished (S13).

Step 4 (S15): Otherwise, the completion request completes at least one message in that channel and RNIC updates the connection context as follows:

• *Context[Ch#]::LastCompletedMsgSN* is updated with *Context[Ch#]::NextToCompleteMsgSN*; and

• *Context[Ch#]::NextToCompleteMsgSN* is updated with the last SN of the next message in the channel. The last SN of the next message can be retrieved in one of following ways:

(1) Explicitly in the *MessageDescriptor::MessageLengh/SN* field, when the *MessageDescriptor::SN/MsgLengthBit* indicates that this field carries the Sequence Number;

(2) Implicitly, using the *MessageDescriptor::MessageLength* as a size of Application payload, and adding the protocol headers, footers and control information (DDP headers, markers, padding, CRC); and

(3) Invalid - when the channel does not have the next transmitted message. This includes the case when no messages were posted, and the case when the message has been posted, but not entirely transmitted.

[0043] Note that if a completed message is not an RDMA Read Request, then RNIC performs completion of the RDMA operation, the details of which are not relevant to the subject of this invention.

[0044] In the case of an RDMA Read Request, RNIC waits for the RDMA Read Response to perform completion, and sets the *Context[Ch#]::PendingReadRequest* bit.

Step 5: Go back to the step 1.

RDMA Read Request Completion

[0045] RNIC completes posted RDMA requests upon receiving a TCP Ack covering the request. An RDMA Read Request is an exceptional case that is completed when the corresponding RDMA Read Response has been received.

[0046] RDMA ordering rules require RDMA requests to be completed in the order they have been sent. This means that RNIC needs to wait for reception of the RDMA Read Response before completing RDMA requests following an RDMA Read Request. Processing of a received RDMA Read Response is shown in a flow chart in Figure 6, and involves the following steps:

Step 1 (S20): If necessary, perform completion of RDMA Requests preceding the RDMA Read Request.

This is a very rare case, and occurs when completion requests have not been processed until reception of RDMA Read Response. Thus, RNIC needs first to complete all requests preceding the RDMA Read Request. This case is indicated by non-zero

Context[Ch#]::PendingReadRequestsCount and clear

Context[Ch#]::PendingReadRequest bit.

Step 2 (S21): Perform Completion of the RDMA Read Request itself.

RNIC performs completion of the RDMA Read operation; the details of are not relevant to the invention.

Step 3 (S22): Perform completion of any RDMA requests following a completed RDMA Read Request.

This process is described above under the section "Ack Completion."

Step 4 (S23): Repeat steps 2-3 N number of times, wherein N is the number of completed RDMA Read Requests defined by *Context[Ch#]::CompletedReadRequestsNum.*

An example of these processes is also depicted in Figure 7.

Retransmit Request Flow

[0047] Processing of a retransmit request involves addressing the following issues:

Issue 1: RNIC needs to process all pending completion requests for this connection. Therefore, if a connection has an outstanding completion request, then the RNIC is

requested to process this request. Processing of the retransmit request is resumed only after completion processing for the connection has been finished.

Issue 2: Retransmit can be requested either due to a fast-retransmit mechanism, or due to a timeout. Retransmit due to a timeout involves the retransmission of all transmitted, but not completed, TCP segments. Retransmit due to a fast-retransmit mechanism involves retransmission of the first not completed TCP segment only. RNIC uses different retransmit request types. Processing of the retransmit request depends on the request type, which are described below.

Issue 3: One of crucial parts of retransmit request processing is the location of the segment to retransmit in the RequestOut or ResponseOut descriptor lists. This process is described below.

[0048] Retransmit does not necessarily preserve boundaries of transmitted segments. The location from which retransmit should be started is indicated by *LastAckedSN*. Socket and iSCSI are not sensitive to changes of the transmitted segment boundaries. iSCSI assumes that TCP does not submit requests to perform retransmit for the not-transmitted data. This allows iSCSI Data Processing logic to assume that during retransmit, all inbound Immediate Command descriptors carry a valid iSCSI CRC, calculated during transmit.

[0049] RDMA preserves boundaries of transmitted DDP segments, and thus may start retransmit from a location other than *LastAckedSN*.

Retransmit Request Types

[0050] As shown in Figure 8, TCP posts three types of retransmit requests:

(1) Retransmit due to fast-retransmit;

(2) Start retransmit due to timeout; and

(3) Proceed timeout retransmit.

[0051] All types of retransmit requests involve retransmission of a single TCP segment. RNIC keeps boundaries of completed, waiting for completion, and waiting for transmit messages, using NextToCompletion and NextToSend pointers, per RequestOut and ResponseOut channels. RNIC also keeps a number of posted and not transmitted messages (*Context[Ch#]::TotalPostedMsgCount*), and a number of transmitted messages waiting for completion (*Context[Ch#]::TotalPendingMsgCount*) per connection. Note that both *Context[Ch#]::TotalPostedMsgCount* and *Context[Ch#]::TotalPendingMsgCount* are kept for each connection, and are used for both ResponseOut and RequestOut channels. In case of fast-retransmit, RNIC just retransmits the first not completed segment (i.e., the TCP segment inside message referred by NextToComplete), without updating any of the pointers or counter mentioned above. The channel to retransmit from is selected as described below.

[0052] In the case of start retransmit due to timeout, RNIC updates the NextToSend pointer of one or both channels to refer to the first not completed TCP segment in that channel, inside message referred by NextToComplete (N2C), and retransmits a single TCP segment from this location. The *Context[Ch#]::TotalPostedMsgCount* and *Contrext[Ch#]::TotalPendingMsgCount* are updated respectively. The NextToSend (N2S) channel pointer is advanced to refer to the

next segment to be retransmitted (or transmitted) in that channel. In case of a proceed timeout retransmit, RNIC retransmits a single TCP segment from the location referred by NextToSend, and updates NextToSend and *Context[Ch#]::TotalPendingMsgCount* respectively. Note that TCP does not request retransmission of TCP data that has not been transmitted.

Locating of Segment to Retransmit

[0053] The location of the TCP segment to retransmit is affected by having two channels, RequestOut and ResponseOut, holding transmitted messages. The location of the segment to retransmit consists of several steps, outlined below:

Step 1: Finding a first not-completed message in the RequestOut channel.

[0054] After this step, each channel (RequestOut and ResponseOut) has a candidate message. This step is relevant only when a channel waits for completion of a pending RDMA Read Request (i.e., *Context[Ch#]::PendingReadRequest* bit set), and therefore is only relevant for the RequestOut channel.

[0055] This involves a process very similar to the regular completion walk-through of the channel messages process described above, in the discussion of calculation options under the section entitled "Completion Flow." The goal of this process is to find a message having a last sequence number that exceeds a *Context[Ch#]::LastAckedSN.* This message potentially carries a segment to retransmit. As shown in Figure 9, this is message #5 in Example A, and is message #8 in Example B. The walk through process involves calculation of the last sequence number of the message. Note an important

difference of this walk through for the RequestOut descriptors versus the walk through described for the regular completion operation due to reception of a TCP Ack as described above. The difference is that in a walk through due to retransmit, RNIC ignores RDMA ordering rules, and treats an RDMA Read Request as a uni-directional RDMA operation (like Send or RDMA Write). This is done since the goal is to get to the first RDMA message that was not successfully transmitted by TCP.

[0056] In the example described above in Figure 9 for the case of a completion due to reception of RDMA Read Response, the regular completion handling process would end with ReqOut::NCSN = 100, ReqOut::LCSN = 0, RespOut::NCSN = 600, RespOut::LCSN = 300 for both examples. (Note that RespOut already points to the candidate WQE). In the ReqOut channel for both examples, the completion process would stop on the first pending RDMA Read Request (which is the first WQE# in these examples). The difference between the two examples is that an AckSN of the last received Ack would be 450 for Example A, and 550 for Example B. This is the initial condition for the retransmit operation. The retransmit operation starts with the walk though in the ReqOut channel. In Example A, RNIC walks through WQE#1 and WQE#4, and reaches WQE#5, which ends with a larger SN (500) than LastAckedSN (450). In Example B, the walk through stops at WQE#8. The NCSN and LCSN are then updated respectively for each example.

[0057] Note that the walk-through process is relevant to a RequestOut channel having a pending RDMA Read Request. Otherwise, in the case of no Read Request, or a ResponseOut channel, the next-to-complete (N2C) always points to the message candidate. Note that prior to execution of a retransmit request, regular completion must

be performed for both channels, just to make sure that there are no pending completion operations.

Step 2: Selecting between two candidate messages (one belonging to each of the RequestOut and ResponseOut channels), a message carrying the segment to retransmit.

[0058] To select the message holding the segment to retransmit, RNIC compares *Context[Ch#]::ReqOutNextToCompleteMsgSN* with *Context[Ch#]::RespOutNextToCompleteMsgSN*. The smaller value belongs to the channel that holds a message with segment to retransmit.

[0059] Figure 9 demonstrates two examples. In example A (ReqOut::NCSN = 500, RespOut::NCSN = 600), the selected message belongs to the RequestOut channel, i.e., message #5. In example B (ReqOut::NCSN = 800, RespOut::NCSN = 600), the selected message belongs to the ResponseOut channel, i.e., message #6.

Step 3: Finding the data buffer belonging to the message to start retransmit from.

[0060] This step, after message selection, determines the location of the Pointer descriptor referring to the beginning of the segment to retransmit. To find the Pointer descriptor, RNIC needs to calculate a starting sequence number of the message. Note that the starting SN of the message can be utilized to help find the buffer offset inside the message, and this offset can be used to find the buffer to retransmit from. The last sequence number of the message is known, and is kept in connection context (*Context[Ch#]::ReqOutNextToCompleteMsgSN* or *Context[Ch#]::RespOutNextToCompleteMsgSN*).

[0061] To calculate the starting sequence number of the message, RNIC compares the *Context[Ch#]::ReqOutLastCompletedMsgSN* with *Context[Ch#]::RespOutLastCompletedMsgSN*. The larger value indicates a starting sequence number of the message, more accurately, the starting sequence number of the message equals to max(*Context[Ch#]::ReqOutLastCompletedMsgSN*, *Context[Ch#]::RespOutLastCompletedMsgSN*) + 1.

[0062] As shown in Figure 9, in Example A, the starting sequence number is 401, and in Example B, the starting sequence number is 501. Once the starting sequence number of the message is found, RNIC walks through the Pointer Descriptors and looks for the Pointer Descriptor holding the beginning of the segment to retransmit. Note that generally, *LastAckedSN* indicates the beginning of the segment to retransmit.

[0063] It is understood that the systems, functions, mechanisms, methods, engines and modules described herein can be implemented in hardware, software, or a combination of hardware and software. They may be implemented by any type of computer system or other apparatus adapted for carrying out the methods described herein. A typical combination of hardware and software could be a general-purpose computer system with a computer program that, when loaded and executed, controls the computer system such that it carries out the methods described herein. Alternatively, a specific use computer, containing specialized hardware for carrying out one or more of the functional tasks of the invention could be utilized. The present invention can also be embedded in a computer program product, which comprises all the features enabling the implementation of the methods and functions described herein, and which - when loaded in a computer system - is able to carry out these methods and functions. Computer

program, software program, program, program product, or software, in the present context mean any expression, in any language, code or notation, of a set of instructions intended to cause a system having an information processing capability to perform a particular function either directly or after either or both of the following: (a) conversion to another language, code or notation; and/or (b) reproduction in a different material form.

[0064] The foregoing description of the preferred embodiments of the invention has been presented for purposes of illustration and description. They are not intended to be exhaustive or to limit the invention to the precise form disclosed, and obviously many modifications and variations are possible in light of the above teachings. Such modifications and variations that are apparent to a person skilled in the art are intended to be included within the scope of this invention as defined by the accompanying claims.